LAB 4
Waveform Generator Project

B EE 425: Lab Session AA (Winter 2021)

Austin Gilbert - 1737454
Adrian - 1978456
Carol Kao - 1524575

Lab Due: March 15, 2020 @ 11:59 PM

Abstract

The objective of this lab is to successfully assemble, test, and program the TIVA TM4C123GXL Launchpad attached to the BEE425L21 lab-specific PCB and 4x4 Keypad to run a simple amplitude-adjustable, frequency-adjustable waveform generator. Using the Keil MDK integrated development environment, we were able to run high-level C and load it onto the microcontroller via a USB cable to edit, assemble, and test our program. Additionally, we used a Digilent Analog Discovery 2 Module (AD2) to provide power to the PCB and read/drive all inputs and outputs.

Introduction

The BEE425L21 board contains a Pulse Width Modulator, DAC0808 Analog to Digital Converter, two potentiometers, and two op-amps that connect directly to the TIVA TM4C123GXL Launchpad, 4x4 keypad, and Analog Discovery 2 Module. With everything connected, the device acts to generate analog AC and DC input, analog output, digital conversion output, and pulse width modulation with low pass filter implementation. This allows University of Washington students to work on several different lab projects and procedures for this course while giving them the opportunity to work directly on non-simulated electronics while in quarantine.

On the software side, we were given Quality Control Test Software which defines all needed ports of the two attached boards, keypad control code, Systick timing integration, ADC integration, and DAC integration. This software is then split into two modes, each activated by pressing SW1 or SW2 on the attached TIVA board. These two modes allow the user to switch between a wave mode that captures ADC input to be read via the AD2 and a scan mode which scans the keypad inputs and deciphers them into hexadecimal values.

This project was split into three milestones: software design, first code draft, and final code draft with specific test criteria. Using bits and pieces from previous labs and the QC Test code, we were able to specify a design for our waveform generator in Milestone 1 and develop our code around that initial design.

Procedure I

Assembling the Lab Kit

We began this project by assembling the lab kit. We first attached the TIVA Launchpad to the BEE425L21 board via its J1 and J2 ports by aligning the pins and snapping them all on at once. We then hooked up the 4x4 Keypad to the back of the PCB via its J4 port, again snapping all pins on at once. After this, we began the arduous process of attaching each AD2 flywire to the BEE425L21's J3 port, looking to the wiring table in the lab manual to ensure each pin was

connected correctly. Once all pins were successfully connected, we then attached the USB-to-micro-USB cables from the AD2 and TIVA board to our PCs (*Figure 1*).

After downloading the board-specific Quality Control test program via a zip file on Canvas titled “BEE425L21 QC - old,” we opened the “Lab2” µVision4 project file. As per the lab manual instructions of the previous lab, we chose the target to be the Texas Instruments TM4C123GH6PM and configured the flash tools. After setting the wave mode variable, “WaveM” equal to zero and the low pass filter mode, “LPFM” equal to seven, and building all targets, we flashed and reset the TIVA module. The first portion of our C program is shown in *Figure 2*.

After downloading the “BEE 425 Lab WFW” Waveforms file and turning on both 5 V supplies, the scope, and the logic analyzer, we were able to begin altering our code.

Figure 1: Assembled Lab Kit

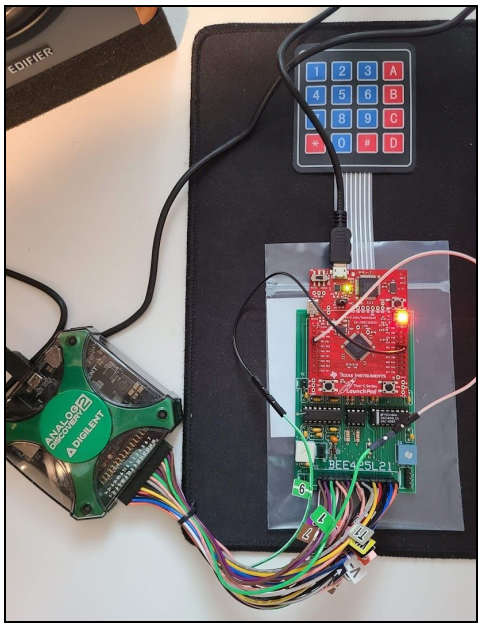


Figure 2: Quality Control Test Program (Lines 1-69)

```
1  // BEE 425 AA, Winter 2021
2  // Austin Gilbert, Adrian, & Carol Kao, 02/22/21
3  // Modified from Valvano et al, UTexas & Joseph Decuir, UWashington
4  // BEE425L21 Part 1: Assembling & Testing the Lab Kit
5  // main.c
6
7  // 1.0 DESCRIPTION:
8  // Two QC modes:
9  // 1) SW1 Waveform mode, with limited key scanning (default)
10 // 1.1) Waveform mode indication with RED LED
11 // 1.2) compile time settings:
12 // 1.2.1) WaveM: 0 = ramp (16 samples); 1=square; 2=sine (12 samples); 3=echo
13 // WaveM 0-2: software generates waves; WaveM 3: copy top 8-bits from ADC to PB
14 // 1.2.2) LPFM: 0-3 off; 4=72kHz; 5=7.2kHz; 6=720Hz; 72Hz
15 // 2) SW2 Keyscanning
16 // 2.1) Keyscan mode indication with GREEN LED
17 // 2.2) Key scanning on PC7-4 and PD3-0
18 // 2.3) PB7-4 off; 4-bit key code output on PB3-0
19 // 2.3) Active key indicated (transiently) on BLUE LED
20
21 // 2.0 Pre-processor Directives Section
22 // Constant declarations to access port registers using symbolic names
23 // modified to include TM4C123GH6PM.h & system_TM4C123.h definitions
24 #include "TM4C123GH6PM.h"       // Keil seems to ignore them
25 #include "system_TM4C123.h"     // ditto
26 #include "TExaS.h"
27 // master port clock
28 #define SYSCTL_RCGC2_R          (*((volatile unsigned long *)0x400FE108))
29 // GPIO Port F
30 #define GPIO_PORTF_DATA_R       (*((volatile unsigned long *)0x400253FC))
31 #define GPIO_PORTF_DIR_R        (*((volatile unsigned long *)0x40025400))
32 #define GPIO_PORTF_AFSEL_R      (*((volatile unsigned long *)0x40025420))
33 #define GPIO_PORTF_PUR_R        (*((volatile unsigned long *)0x40025510))
34 #define GPIO_PORTF_DEN_R        (*((volatile unsigned long *)0x4002551C))
35 #define GPIO_PORTF_LOCK_R       (*((volatile unsigned long *)0x40025520))
36 #define GPIO_PORTF_CR_R         (*((volatile unsigned long *)0x40025524))
37 #define GPIO_PORTF_AMSEL_R      (*((volatile unsigned long *)0x40025528))
38 #define GPIO_PORTF_PCTL_R       (*((volatile unsigned long *)0x4002552C))
39 // NVIC = Systick
40 #define NVIC_ST_RELOAD_R        (*((volatile unsigned long *)0xE000E014))
41 #define NVIC_ST_CTRL_R          (*((volatile unsigned long *)0xE000E010))
42 // GPIO Port E
43 #define GPIO_PORTE_DATA_R       (*((volatile unsigned long *)0x400243FC))
44 #define GPIO_PORTE_DIR_R        (*((volatile unsigned long *)0x40024400))
45 #define GPIO_PORTE_AFSEL_R      (*((volatile unsigned long *)0x40024420))
46 #define GPIO_PORTE_PUR_R        (*((volatile unsigned long *)0x40024510))
47 #define GPIO_PORTE_DEN_R        (*((volatile unsigned long *)0x4002451C))
48 #define GPIO_PORTE_CR_R         (*((volatile unsigned long *)0x40024524))
49 #define GPIO_PORTE_AMSEL_R      (*((volatile unsigned long *)0x40024528))
50 #define GPIO_PORTE_PCTL_R       (*((volatile unsigned long *)0x4002452C))
51 // ADC AIN0 (PE3)
52 #define SYSCTL_RCGCADC_R        (*((volatile unsigned long *)0x400FE638))
53 #define ADC0_ACTSS_R            (*((volatile unsigned long *)0x40038000))
54 #define ADC0_EMUX_R             (*((volatile unsigned long *)0x40038014))
55 #define ADC0_SSMUX3_R           (*((volatile unsigned long *)0x400380A0))
56 #define ADC0_SSCTL3_R           (*((volatile unsigned long *)0x400380A4))
57 #define ADC0_PSSI_R             (*((volatile unsigned long *)0x40038028))
58 #define ADC0_SSFIFO3_R          (*((volatile unsigned long *)0x400380A8))
59 #define ADC0_IM_R               (*((volatile unsigned long *)0x40038008))
60 #define ADC0_ISC_R              (*((volatile unsigned long *)0x4003800C))
61 #define ADC0_RIS_R              (*((volatile unsigned long *)0x40038004))
62 #define ADC0_SSPRI_R            (*((volatile unsigned long *)0x40038020))
63 // GPIO Port D
64 #define GPIO_PORTD_DATA_R       (*((volatile unsigned long *)0x400073FC))
65 #define GPIO_PORTD_DIR_R        (*((volatile unsigned long *)0x40007400))
66 #define GPIO_PORTD_AFSEL_R      (*((volatile unsigned long *)0x40007420))
67 #define GPIO_PORTD_PUR_R        (*((volatile unsigned long *)0x40007510))
68 #define GPIO_PORTD_DEN_R        (*((volatile unsigned long *)0x4000751C))
69 #define GPIO_PORTD_CR_R         (*((volatile unsigned long *)0x40007524))
```

Procedure II

Software Design

Our software design is implemented using the following user interface. Upon flashing and resetting the TIVA Launchpad, the user selects a number on the 4x4 Keypad which indicates the wave type, then presses star. The LED turns red when in this mode to identify that the user is in Keypad-scanning mode. Our program treats star as a terminating key, taking in the digit pressed before star and changing from key-scanning mode to a specific waveform mode if the digit is one of the following:

- **Key 1 -** Sine Wave (variable amplitude, 83.3 Hz default frequency, variable LPF)
- **Key 2 -** Ramp Wave (variable amplitude, 62.5 Hz default frequency, variable LPF)
- **Key 3 -** Triangle Wave (variable amplitude, 83.3 Hz default frequency, variable LPF)
- **Key 4 -** Square Wave (variable amplitude, 500 Hz default frequency, no LPF)

If the digit pressed is not one of these keys, our program resets the key input and flashes the red LED to tell the user to try again. Once the user input is correct, our program changes to waveform mode, activating SW1 and SW2 as inputs and turning the LED green. SW1 is activated by default in this mode, allowing the user to alter the potentiometer value to change the amplitude of the wave from 32 mV to 8 V peak-to-peak. Pressing SW2 allows the user to alter the potentiometer value to change the frequency from 10 Hz to 10 kHz.

We also can control the low pass filter using adaptive controls that will vary the LPF value with the frequency to achieve a smoother output. Since we do not want the square wave output to be smooth, we will turn off the low pass filter for the Square Wave.

The advantage of this specific design is that once the wave is selected, the user can change the amplitude and frequency values in real-time, seeing the waveform output as these values change.

We divided the work for this project by function. In other words, we distributed the Lab 4 Report amongst everyone while the code was finished by Austin.

Procedure III

Software Development

Before altering the code, we started with the QC Test Code used in Lab 3 and organized the sections and comments for easier reading.

We used the sample sine wavetable for our signal shown in *Figure 3* and we created another ramp and triangle sample wavetable, which is calculated by converting decimals to hex digits on the waveform without the values turning into a negative number. For example, we have a

decimal number 128 which is 0x80 in hex, and it is set as the “0” on our waveform, so if we subtract 80 from every point on the wave, we get our signal to set at mid-range.

Figure 3 - Sample wavetable

```
112  // 12 sample signwave table
113  int SinT[] = {0x80,0xC0,0xEE,0xFF,0xEE,0xC0,0x80,0x40,0x12,0x00,0x12,0x40};
114  // 16 sample rampwave table
115 int RampT[] = {0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0,
116                0xB0, 0xC0, 0xD0, 0xE0, 0xF0};
117  // 12 sample trianglewave table
118  int TriT[] = {0x80,0xAB,0xD5,0xFF,0xD5,0xAB,0x80,0x55,0x2B,0x00,0x2B,0x55};
119  int SquareT[] = {0x00, 0xFF};               // 2 sample squarewave table
```

Figure 4 - Creating KeyString and Keypad Scanning Mode

```
223
224         // Create KeyString
225         KeyString = KeyHex | ((LastKeyHex << 4) & 0xF0);
226         Delay();
227         GPIO_PORTB_DATA_R = KeyString & 0xFF; // Output to Port B
228
229         // Pressing Last Key "Star"
230         if ((KeyString & 0xF) == 0xE) {
231           // Pressing First Key 1
232           if (((KeyString >> 4) & 0xF) == 0x1) {
233             WaveM = 1;                      // Activate Sine Wave Mode
234             Mode = 1;                       // Branch to else loop
235
236           // Pressing First Key 2
237           } else if (((KeyString >> 4) & 0xF) == 0x2) {
238             WaveM = 2;                      // Activate Sine Wave Mode
239             Mode = 1;                       // Branch to else loop
240
241           // Pressing First Key 3
242           } else if (((KeyString >> 4) & 0xF) == 0x3) {
243             WaveM = 3;
244             Mode = 1;                       // Branch to else loop
245
246           // Pressing First Key 4
247           } else if (((KeyString >> 4) & 0xF) == 0x4) {
248             WaveM = 4;
249             Mode = 1;                       // Branch to else loop
250
251           // Pressing any other Key
252           } else {
253             int i;
254             NVIC_Init(7999999);             // change timing for flashing
255             // Flash RED LED
256             for (i = 0; i <= 2; ++i) {      // for loop to flash twice
257               GPIO_PORTF_DATA_R = 0x00;
258               Delay();
259               GPIO_PORTF_DATA_R = 0x02;
260               Delay();
261             }
262             NVIC_Init(15999);               // change timing back for key-scanning
263             Mode = 0;                       // Restart Mode 0 loop
264           }
265         }
266       }
```

After designing the code to enter “keypad scanning mode” shown in *Figure 4*, we enabled the user to change the amplitude and frequency by turning the white potentiometer on the PCB. And if the user presses a number other than 1-4, the red LED will blink twice, shown in *Figure 4*, indicating the user to try again. To display the waveforms, we initialize the ports and timing for our DAC and LPFM, we have our LPFM set for a range from 0-3 and the remaining 4-7 will be off. After the user presses a valid number key and generates a waveform of their choice, the program is defaulted to allow the user to alter the potentiometer value changing the amplitude, while the LED is green. They also have the choice of changing the frequency of the waveform,

where they will push the SW2 button allowing a switch to frequency mode where the LED will show a sky blue color.

Figure 5 - Waveform Display Mode

```
// WAVEFORM DISPLAY MODE
////////////////////////////////////////////////////////////////////
} else if (Mode == 1) {
  // Initialize Ports & Timing
  NVIC_Init(time);
  GPIO_PORTB_DATA_R = DAC;              // DAC = Port B Output
  GPIO_PORTE_DATA_R = LPFM;             // LPFM = Port E Output
  SW1 = GPIO_PORTF_DATA_R & 0x10;       // sample port PF4
  SW2 = GPIO_PORTF_DATA_R & 0x01;       // sample port PF0
  GPIO_PORTB_DIR_R = 0xFF;              // restore PB7-PB0 output

  // Initialize Small Mode
  if (time < 799) {
    SM = 1;
  }

  // Initialize Switches
  if (SW1 == 0) AdjustM = 0;
  if (SW2 == 0) AdjustM = 1;

  // Variable LPFM
  if (WaveM == 1 || WaveM == 3) { // Sine or Triangle wave
    if (time <= 133332 && time >= 3366) {
      LPFM = 1;                         // 10 Hz - 396 Hz
    } else if (time < 3366 && time >= 336) {
      LPFM = 2;                         // 3.96 kHz - 396 Hz
    } else if (time < 336 && time >= 132) {
      LPFM = 3;                         // 10 kHz - 3.96 kHz
    }
  } else if (WaveM == 2) { // Ramp Wave
    if (time <= 99999 && time >= 2524) {
      LPFM = 1;                         // 10 Hz - 396 Hz
    } else if (time < 2524 && time >= 252) {
      LPFM = 2;                         // 3.96 kHz - 396 Hz
    } else if (time < 252 && time >= 99) {
      LPFM = 3;                         // 10 kHz - 3.96 kHz
    }
  } else { // WaveM == 4
    LPFM = 7;                           // LPF off
  }
```

For the user to alter the amplitude and frequency values with the potentiometer, we set our output (DAC) as a function for each wave. As for our sine function, we set SinT at the sine wave index where we inputted our sine wave table along with the potentiometer ratio. The function is set up as shown in *Figure 6*, where the waveIn takes in every step of the sine wave index and subtracts the value with 0x80 resulting in our signal sitting at mid-range.

Figure 6 - Code for Current ADC Input

```
304         // Capture Current ADC Input
305         ADC0_PSSI_R |= 8;                       // 7) start a conversion at sequence 3
306         while((ADC0_RIS_R & 8) == 0);           // 8) wait for conversion to complete
307         AIN0 = ADC0_SSFIFO3_R;                  // 9) capture the results
308         pot = ((AIN0 >> 4) & 0xFF) * 1.0;       // get 8-bit potentiometer value
309         potRatio = pot / 255.0;                 // ratio of potentiometer turn
310         ADC0_ISC_R = 8;                         // 10) clear completion flag
311         Delay();                                // wait specified time (1ms)
312
313         // Step All Wave-types
314         Delay();                                // wait specified time (1ms)
315         SWI += 1;                               // step sine wave index
316         if(SWI == 12) SWI = 0;                  // counts up 12 times
317         RWI += 1;                               // step ramp wave index
318         if(RWI == 16) RWI = 0;                  // counts up 16 times
319         TWI+=1;                                 // step triangle wave index
320         if(TWI == 12) TWI = 0;                  // counts up 12 times
321         SQWI += 1;                              // step square wave index
322         if(SQWI == 2) SQWI = 0;                 // counts up 2 times
```

Figure 7 - Function for Amplitude

```
454   // 5.7 Wave output function
455   //    int waveIn      -- the hex input wave value
456   //    double potRatio -- the potentiometer ratio value to use as a percentage
457   //      returns       -- the pot-adjusted output int
458   int Output(int waveIn, double potRatio) {
459     outDouble = (waveIn - 0x80) * 1.0;
460     outDouble *= potRatio;
461     return (int)outDouble + 0x80;
462   }
```

When the user is in amplitude changing mode, the frequency will remain at the defaulted value or the value the user has already altered if they had switched back from SW2 to SW1.

Figure 8 - Amplitude Adjustment Mode

```
324       // AMPLITUDE ADJUSTMENT MODE
325       if (AdjustM == 0) {
326         GPIO_PORTF_DATA_R = 0x08;           // Turn LED GREEN
327         ampRatio = potRatio;                // save amplitude ratio
328
329         // Sine Wave
330         if (WaveM == 1) {
331           if (freqRatio != 0) {             // keep freq value
332             time = Timing(133332, 132, freqRatio);
333           }
334           DAC = Output(SinT[SWI], potRatio); // output sine
335         // Ramp Wave
336         } else if (WaveM == 2) {
337           if (freqRatio != 0) {             // keep freq value
338             time = Timing(99999, 99, freqRatio);
339           }
340           DAC = Output(RampT[RWI], potRatio); // output ramp
341         // Triangle Wave
342         } else if (WaveM == 3) {
343           if (freqRatio != 0) {             // keep freq value
344             time = Timing(133332, 132, freqRatio);
345           }
346           DAC = Output(TriT[TWI], potRatio); // output tri
347         // Square Wave
348         } else if (WaveM == 4) {
349           if (freqRatio != 0) {             // keep freq value
350             time = Timing(799999, 799, freqRatio);
351           }
352           DAC = Output(SquareT[SQWI], potRatio); // output square
353         }
```

When the user enters frequency adjustment mode, the amplitude that was adjusted or defaulted will remain the same as the user is asked to change the frequency value after pushing SW2 as shown in *Figure 7*.

Figure 9 - Frequency Adjustment Mode

```
361         // FREQUENCY ADJUSTMENT MODE
362         } else { // AdjustM = 1
363           GPIO_PORTF_DATA_R = 0x0C;           // Turn LED SKY BLUE
364           freqRatio = potRatio;               // save frequency ratio
365
366           // Sine Wave
367           if (WaveM == 1) {
368             time = Timing(133332, 132, potRatio); // change freq via pot
369             if (SM == 1 && time < 5332) {
370               time = Timing(799999, 799, potRatio);
371               DAC = Output(SquareT[SQWI], ampRatio);
372             } else {
373               SM = 0;
374               DAC = Output(SinT[SWI], ampRatio); // output sine
375             }
376           // Ramp Wave
377           } else if (WaveM == 2) {
378             time = Timing(99999, 99, potRatio); // change freq via pot
379             if (SM == 1 && time < 5332) {
380               time = Timing(799999, 799, potRatio);
381               DAC = Output(SquareT[SQWI], ampRatio);
382             } else {
383               SM = 0;
384               DAC = Output(RampT[SWI], ampRatio); // output ramp
385             }
386           // Triangle Wave
387           } else if (WaveM == 3) {
388             time = Timing(133332, 132, potRatio); // change freq via pot
389             if (SM == 1 && time < 5332) {
390               time = Timing(799999, 799, potRatio);
391               DAC = Output(SquareT[SQWI], ampRatio);
392             } else {
393               SM = 0;
394               DAC = Output(TriT[SWI], ampRatio); // output tri
395             }
396           // Square Wave
397           } else if (WaveM == 4) {
398             time = Timing(799999, 799, potRatio); // change freq via pot
399             DAC = Output(SquareT[SQWI], ampRatio); // output square
400           }
```

When the user is in frequency mode, they are allowed to alter the potentiometer and change the frequency of the signal. From our setup, the user is allowed a range from 10Hz to 10kHz. Although we did run into some trouble with frequencies, as the signal was not able to reach 10kHz and it was originally floating around 2.5kHz. We tried to fix it by cutting down on the signal sample points and setting it to a new function to result in a higher frequency which we were able to reach.

Figure 10 - Frequency Timing Function

```
487  // 5.8 Frequency Timing function
488  //    long max        -- max time value
489  //    long min        -- min time value
490  //    double potRatio -- the potentiometer ratio value to use as a percentage
491  //      returns       -- the pot-adjusted output long
492  int Timing(int max, int min, double potRatio) {
493    max -= min;
494    maxDouble = (double)max;
495    maxDouble *= potRatio;
496    return (int)maxDouble + min;
497  }
```

Test Cases with Screenshots

Test Plan Edited as Work Continues

Test Case I - Following our test criteria from Milestone 1, we successfully implemented the code for our key scanning mode where the keys 1-4 each represent a different waveform and as the user pushes a number from 1-4 followed by the star key, the program registers a specific waveform called and allows the user to alter the amplitude of the signal next.

Test Case I is proven by all the other test cases since each test case uses one of the four modes. This cannot be proven via screenshots since the lab kit reads the key scanned digit press immediately and then shifts into the next mode. Proof can be found in last week's lab recording, where I go through each digit and output each waveform.

Test Case II - All waveforms at 8 V peak-to-peak amplitude, default frequencies, default variable LPF values, and 0 V DC offset.

Figure 11 - Sine Wave Output for 8V peak-to-peak Amplitude

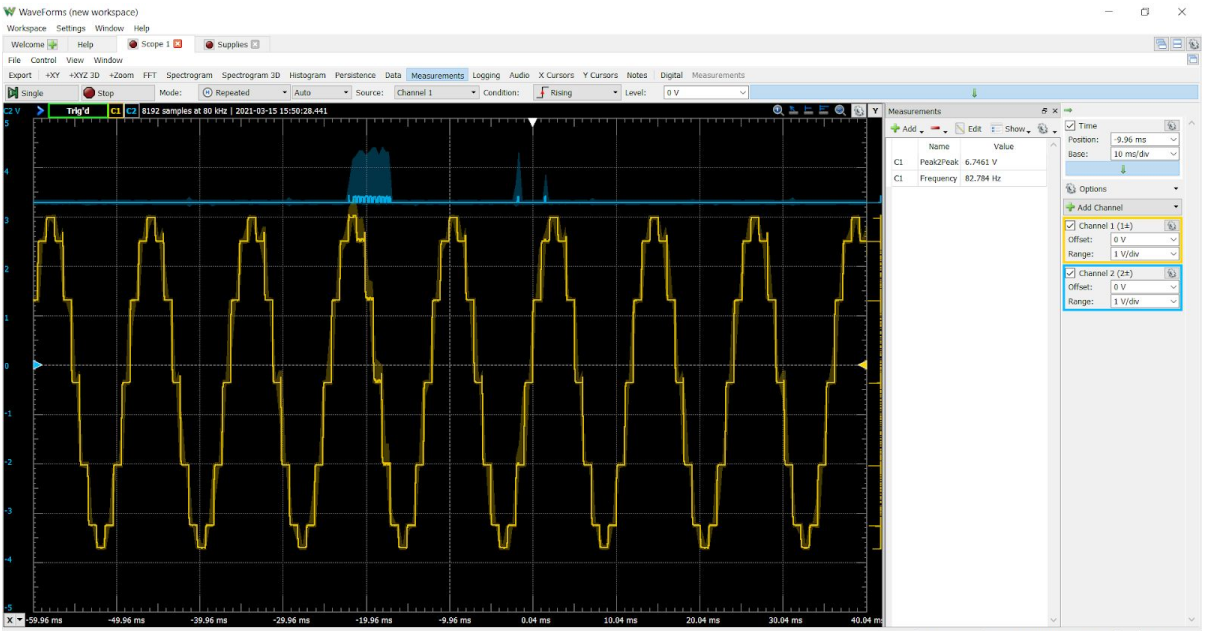


Figure 12 - Ramp Wave Output for 8V peak-to-peak Amplitude

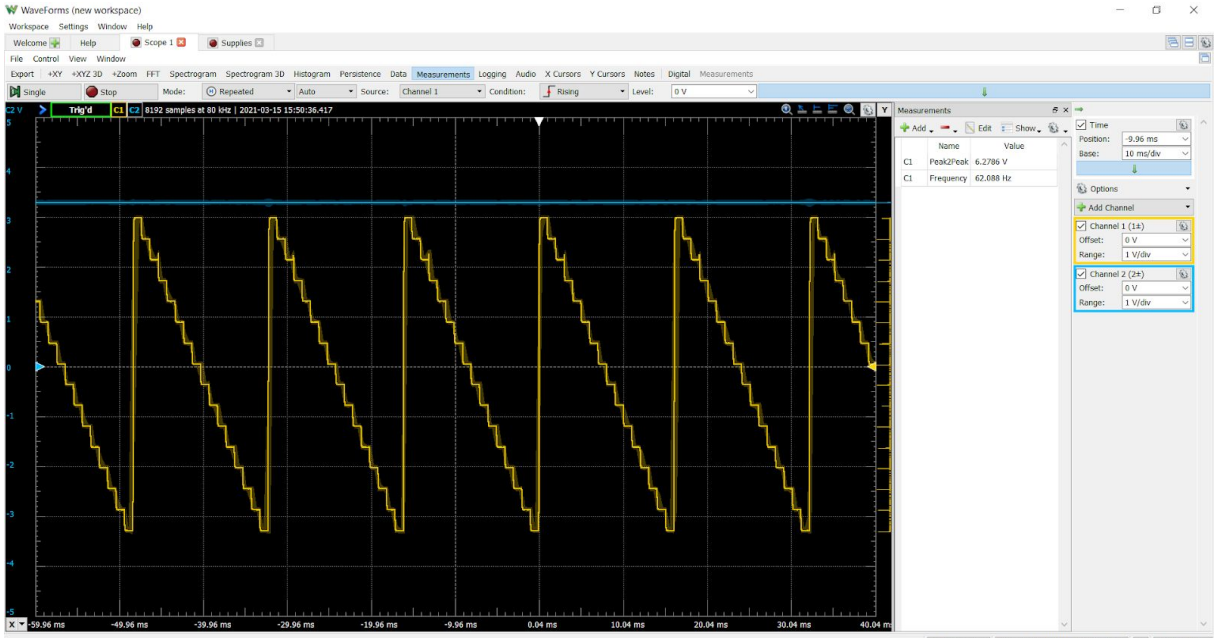


Figure 13 - Triangle Wave Output for 8V peak-to-peak Amplitude

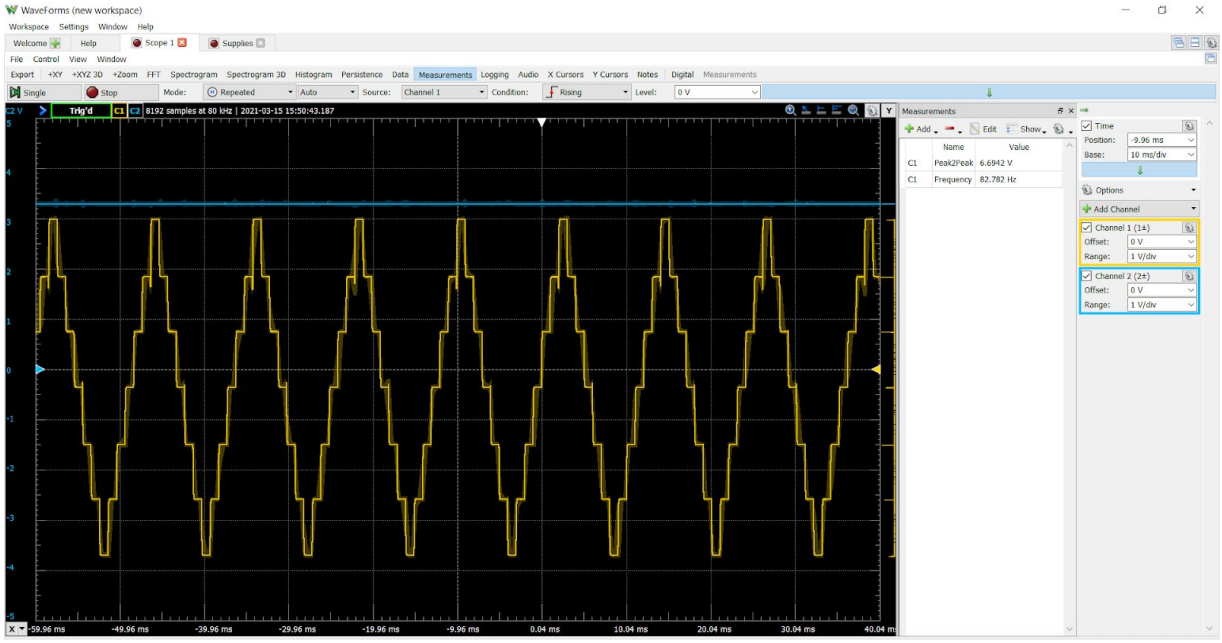
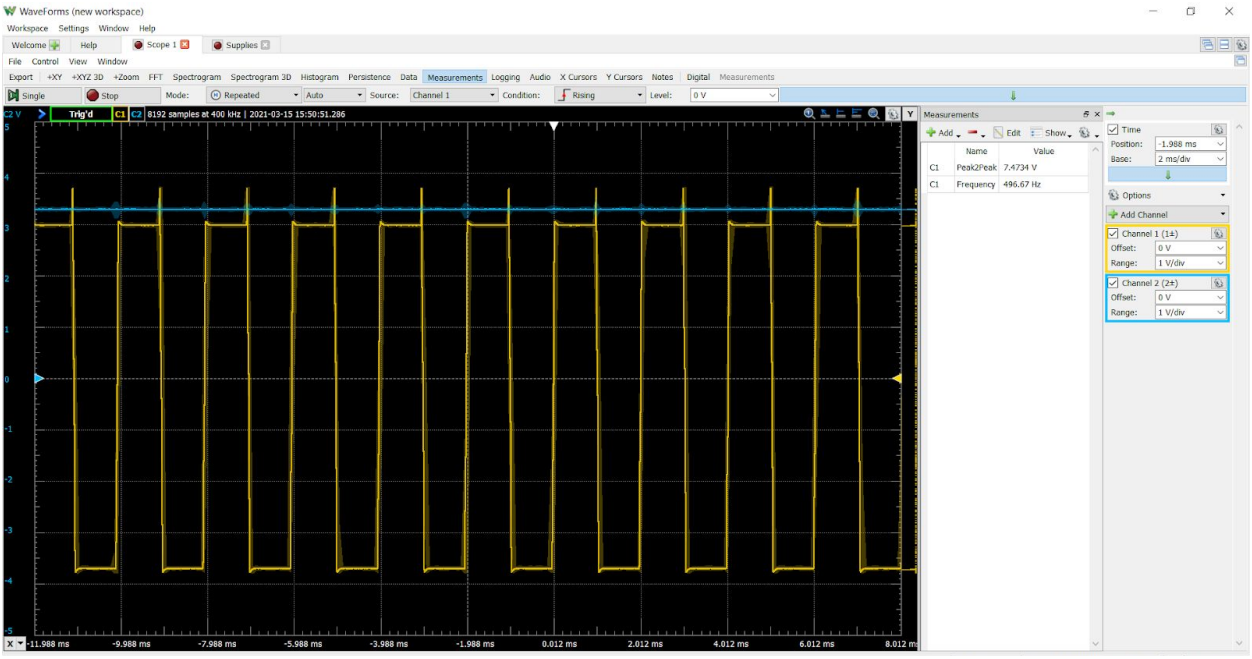


Figure 14 - Square Wave Output for 8V peak-to-peak Amplitude



Test Case III - All waveforms at 4 V peak-to-peak amplitude, 10 kHz frequency, default variable LPF values, and 0 V DC offset.

Figure 15 - Sine Wave Output for 4V peak-to-peak Amplitude

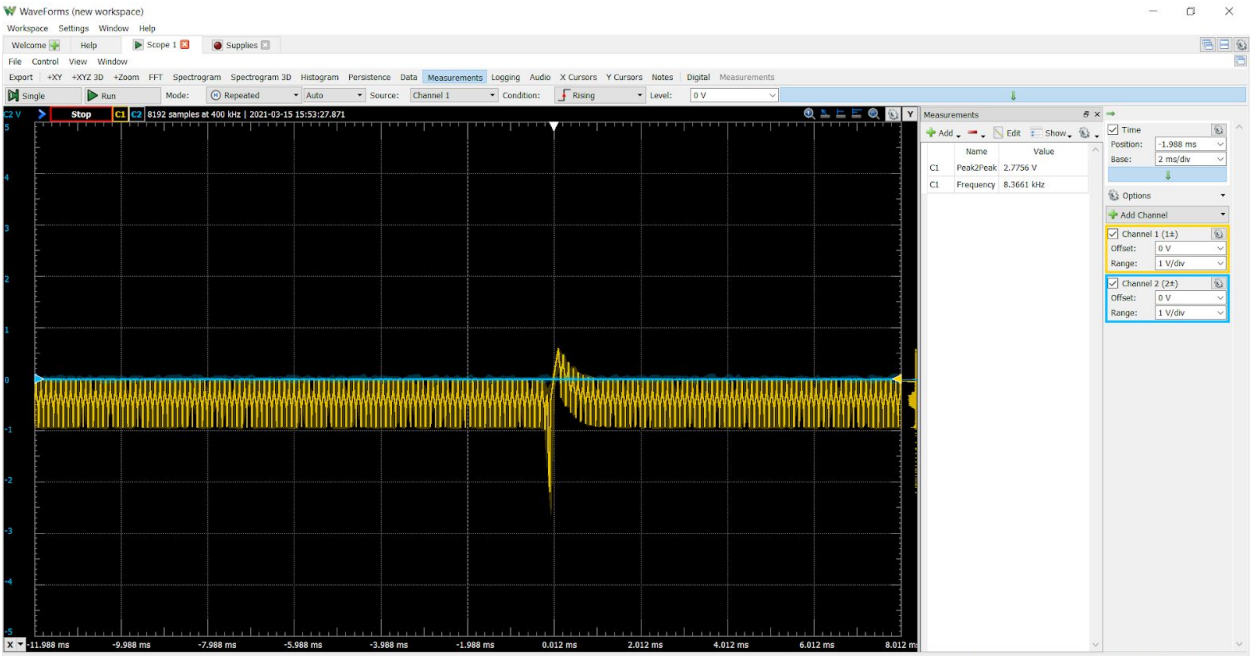


Figure 16 - Ramp Wave Output for 4V peak-to-peak Amplitude

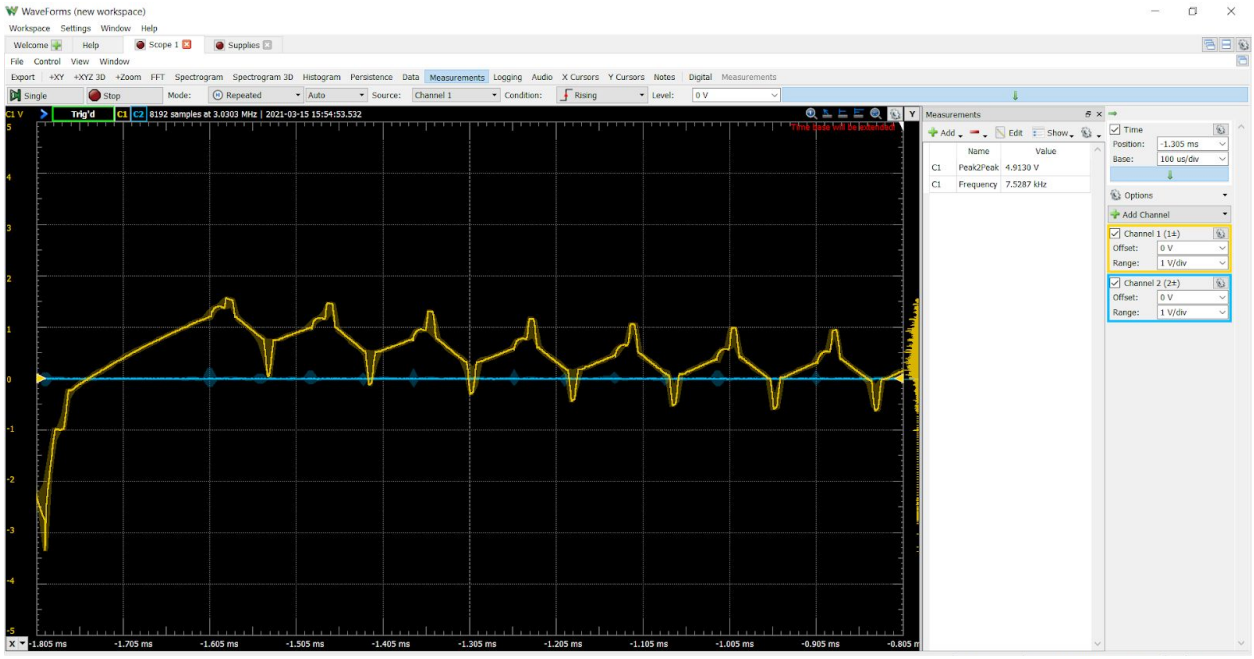


Figure 17 - Triangle Wave Output for 4V peak-to-peak Amplitude

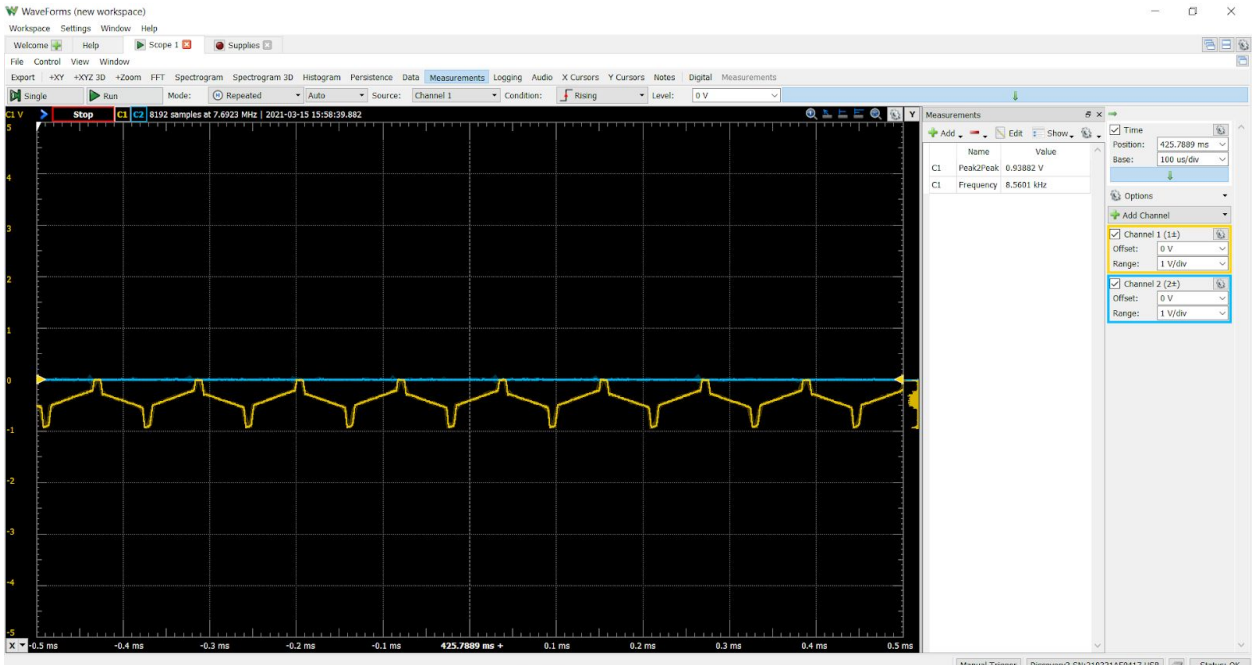
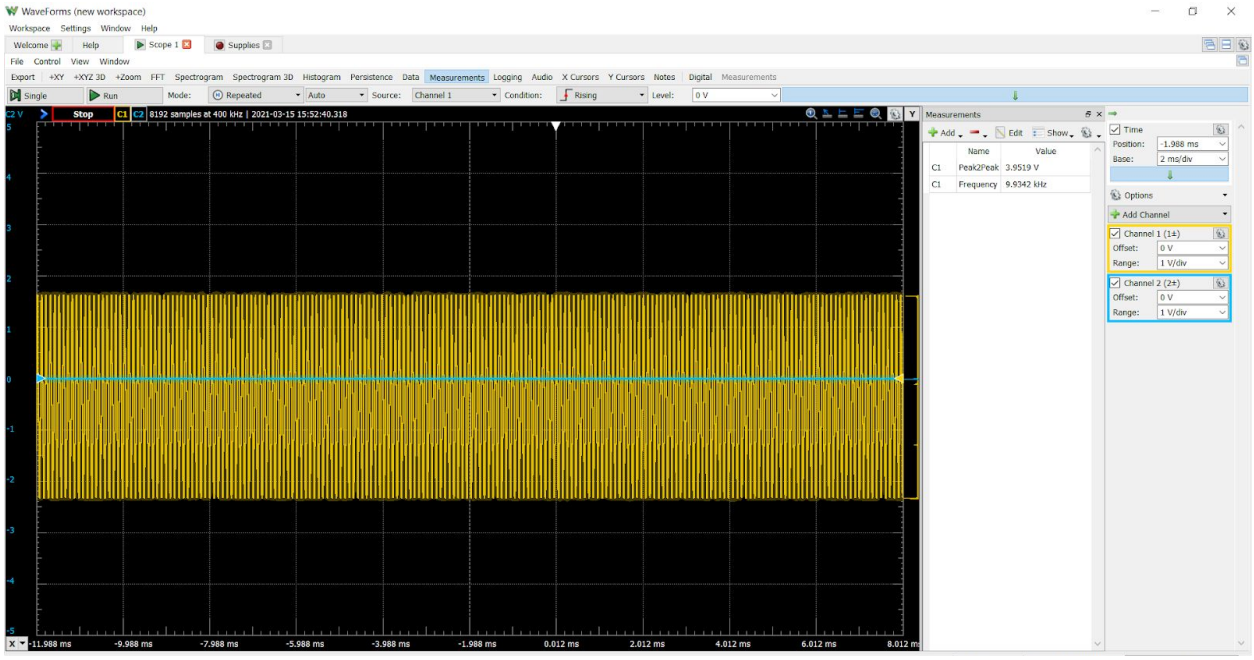


Figure 18 - Square Wave Output for 4V peak-to-peak Amplitude



Test Case IV - All waveforms at 1 V peak-to-peak amplitude, 1 kHz frequency, default variable LPF values, and 0 V DC offset.

Figure 19 - Sine Wave Output for 1V peak-to-peak Amplitude

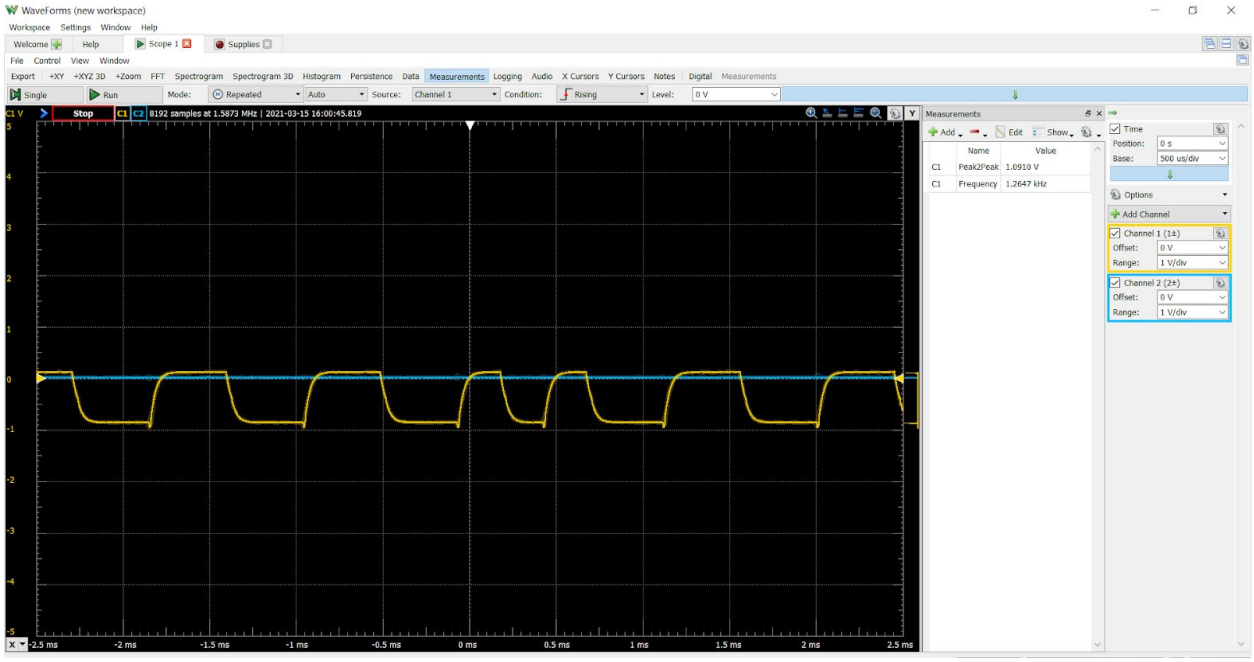


Figure 20 - Ramp Wave Output for 1V peak-to-peak Amplitude

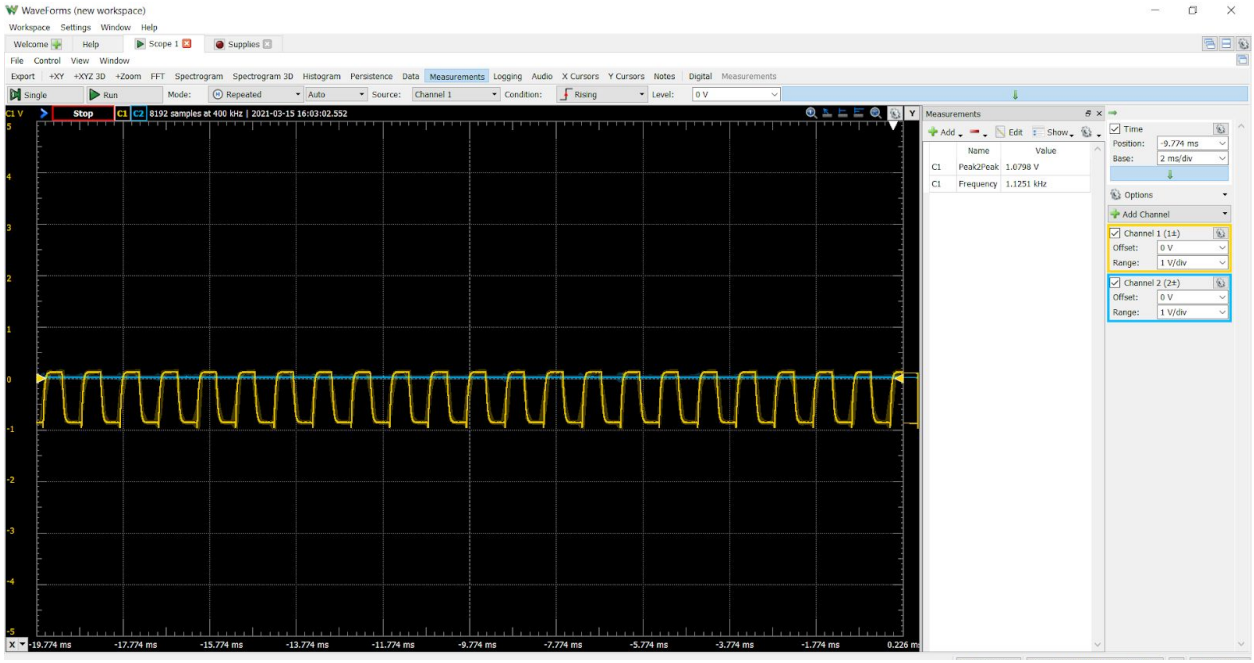


Figure 21 - Triangle Wave Output for 1V peak-to-peak Amplitude

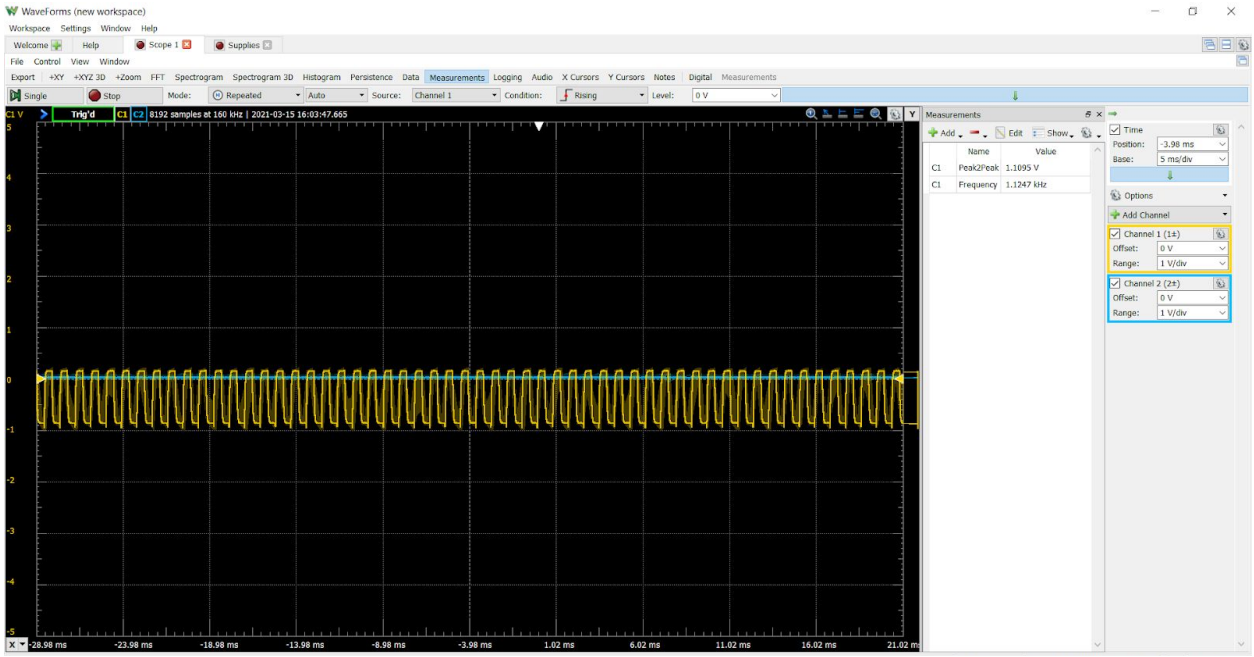
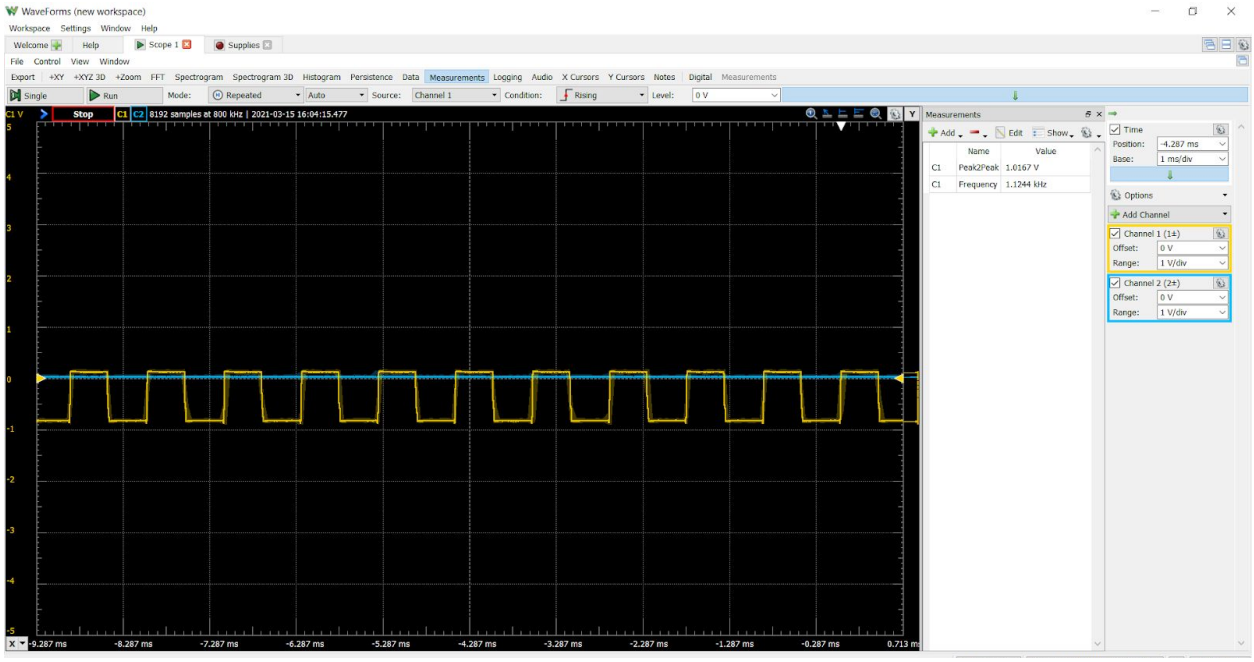


Figure 22 - Square Wave Output for 1V peak-to-peak Amplitude



Test Case V - All waveforms at 32 mV peak-to-peak amplitude, 10 Hz frequency, default variable LPF values, and 0 V DC offset. (The amplitude is the lowest value that we can get by turning the white potentiometer)

Figure 23 - Sine Wave Output for 32mV peak-to-peak Amplitude

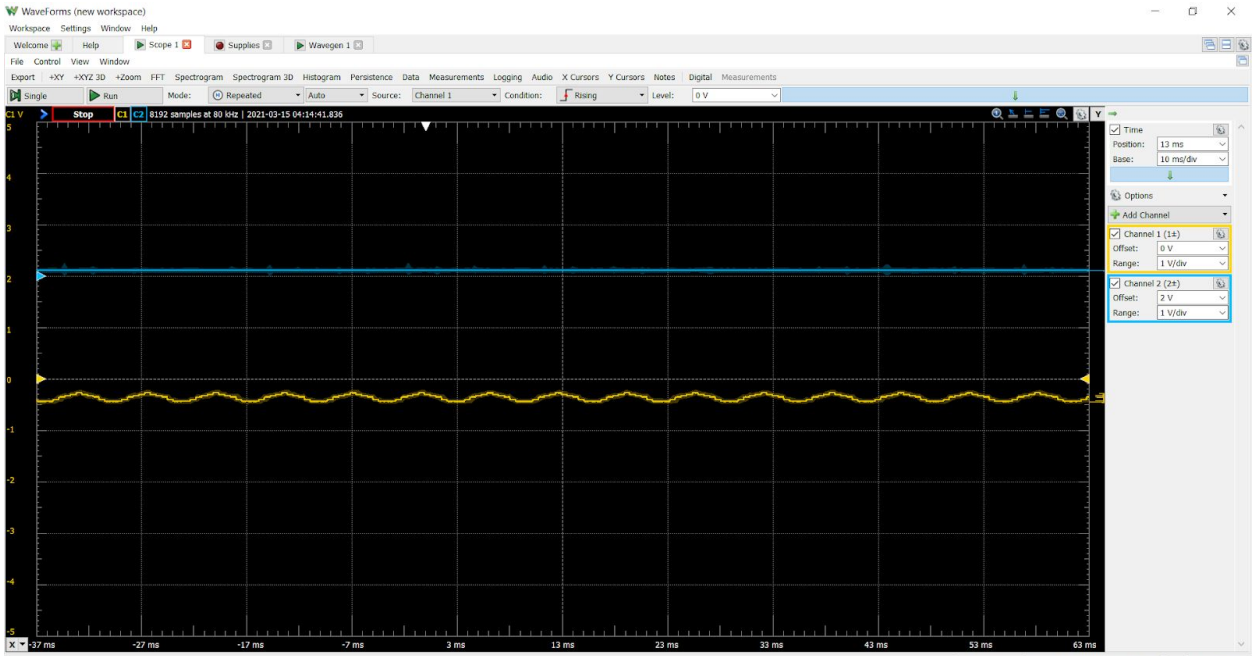


Figure 24 - Ramp Wave Output for 32mV peak-to-peak Amplitude

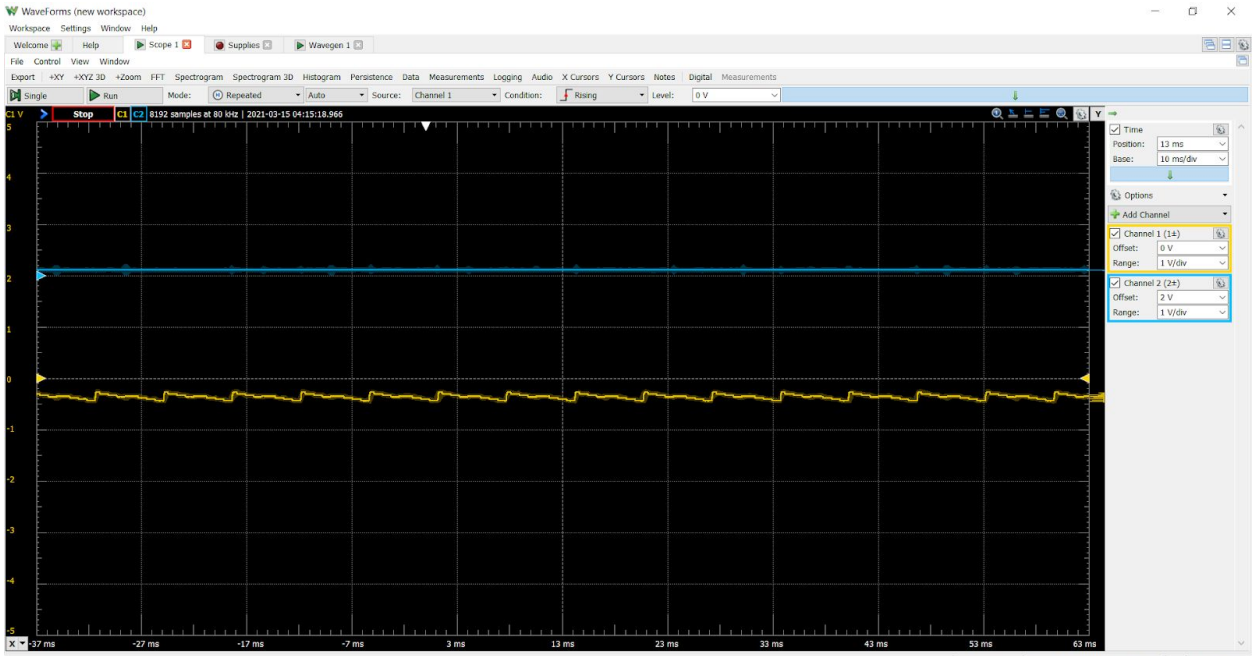


Figure 25 - Triangle Wave Output for 32mV peak-to-peak Amplitude

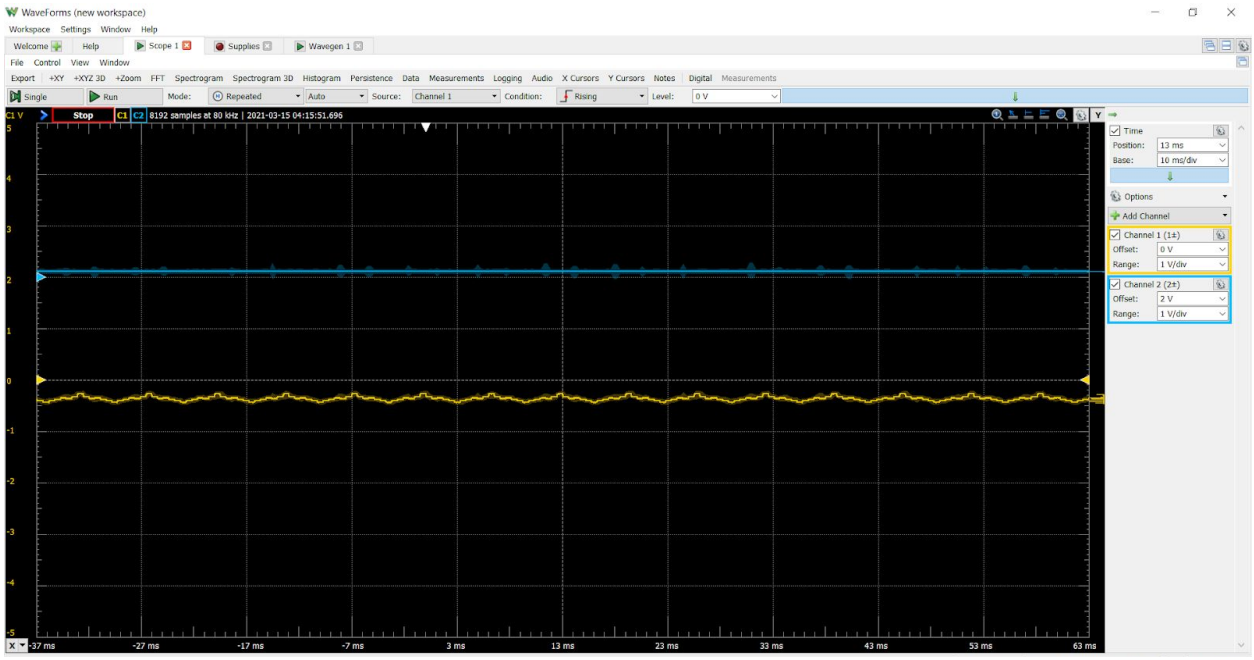
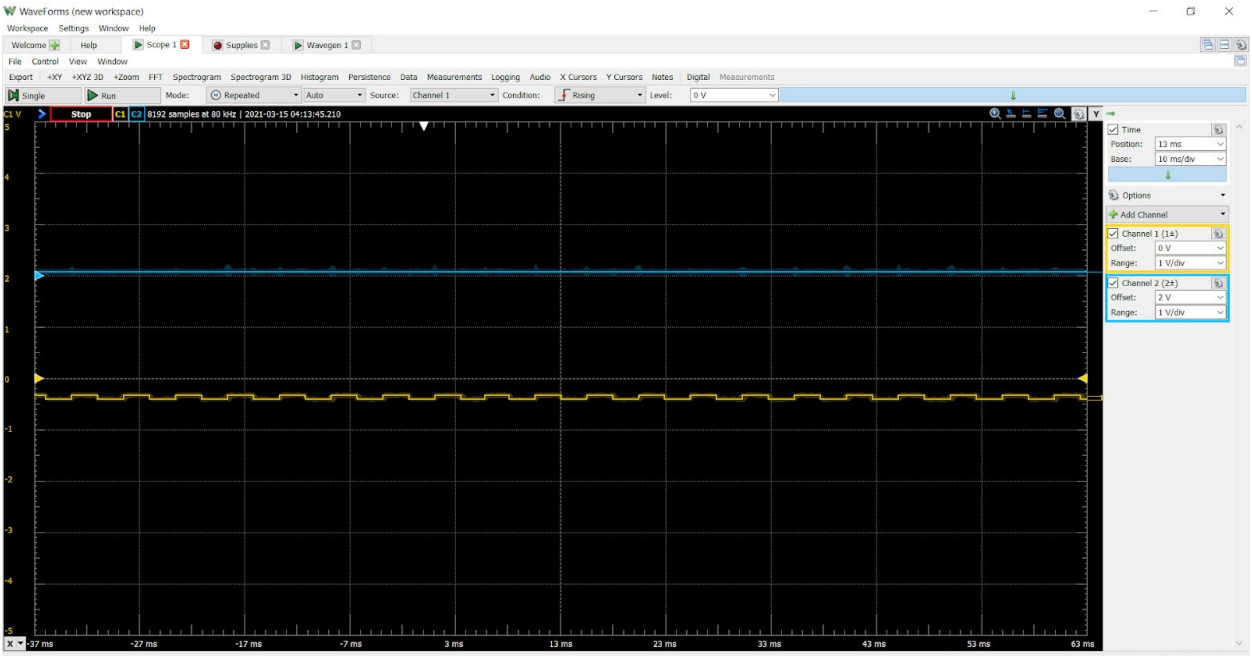


Figure 26 - Square Wave Output for 32mV peak-to-peak Amplitude



Calculations

Math used in the Code and Results

In this lab, Austin created 2 equations that are used to find the frequency value and the NVIC value for each waveform type. The equation that we got is $NVIC = \frac{16*10^6}{f(S_{num})} - 1$, where f is the frequency, and S_{num} is the number of samples. From the equation, we got the maximum and minimum NVIC values for each waveform type.

$$f = \frac{16E6}{S_{num}(NVIC+1)}$$
$$NVIC = \frac{16*10^6}{f(S_{num})} - 1$$

Table 1. Square Wave (2 samples)

NVIC Value	Frequency Value
799999	10Hz
799	10kHz

Table 2. Sine and Triangle Wave (12 Samples)

NVIC Value	Frequency Value

133332	10Hz
132	10kHz
336	3.96kHz
3366	396Hz

Table 3. Ramp Wave (16 Samples)

NVIC Value	Frequency Value
99999	10Hz
99	10kHz
252	3.96kHz
2524	396Hz

Results

Our wave couldn't reach the desired 10KHz frequency even after debugging the code and changing it, we can only get it to 8.5 kHz at most. This means that our Test Case III is not quite right due to how we said the test will be. The amplitude works for every test case except for Test Case I where we could only get around 7V peak-to-peak as the maximum value.

We know our math equation is correct because we tested our equation using our Lab 3 code and it produces the wave with the frequency that we want, and we also know the potentiometer is working from 0 to 0xFF (*see Tables 1,2,3 and 6.1*).

The code shown in *Figure 9* changes the waveform into a square wave if the *time* variable passes 5332. We did this because the maximum frequency that we got before doing this was only around 1.5 kHz for all waveform types.

Figure 27 Using Triangle Waves for smaller samples

```
118  // 12 sample trianglewave table
119  int TriT[] = {0x80,0xAB,0xD5,0xFF,0xD5,0xAB,0x80,0x55,0x2B,0x00,0x2B,0x55};
120  //int TriT[] = {0x80,0xFF,0x80,0x00};
```

We tested using a smaller sample for each waveform and we conclude that when smaller samples are used, a triangle wave used as an example in *Figure 27*, the maximum frequency

increases (we got around 5 kHz for 4 samples of the triangle wave). However, using smaller samples means that the form of the wave will not be as clean as intended.

Figure 28 - Triangle Wave Output for 4 Samples

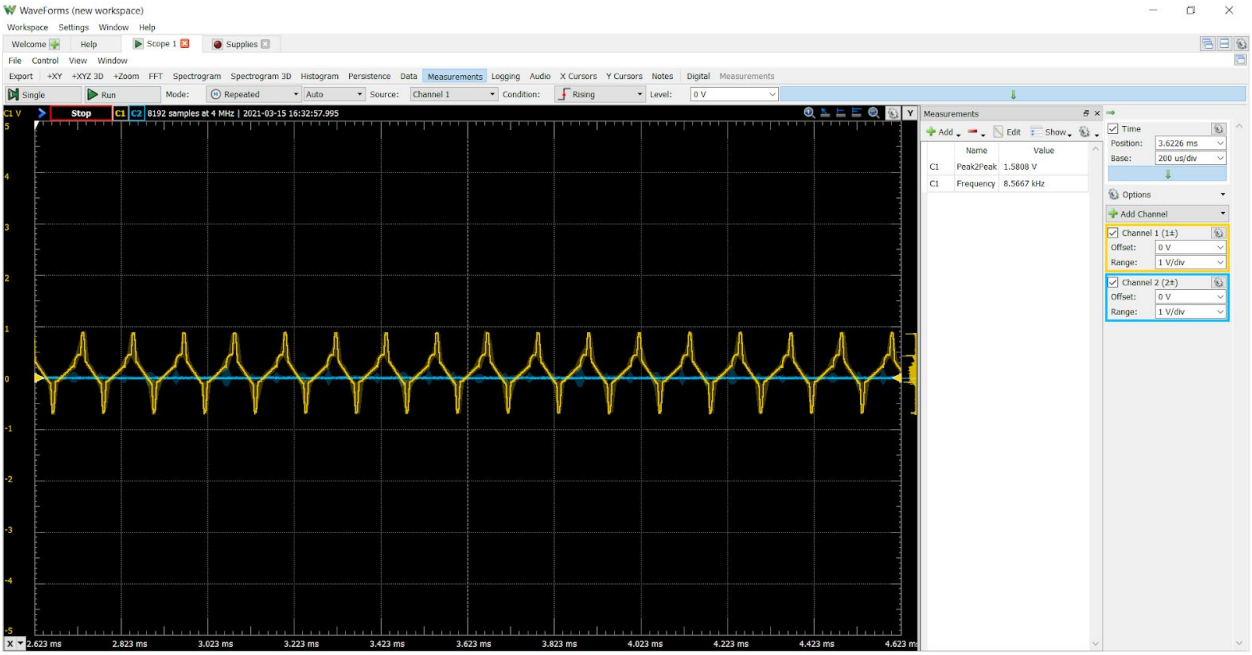


Figure 29 - Potentiometer Output Result (Turned to 0)

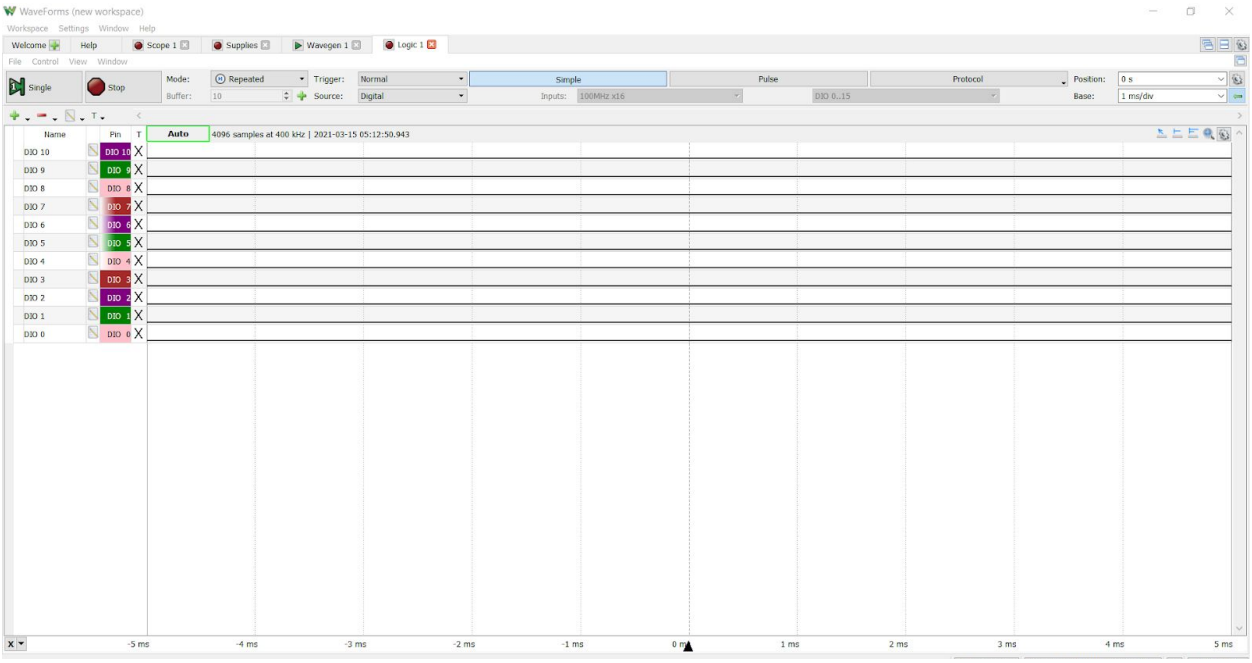
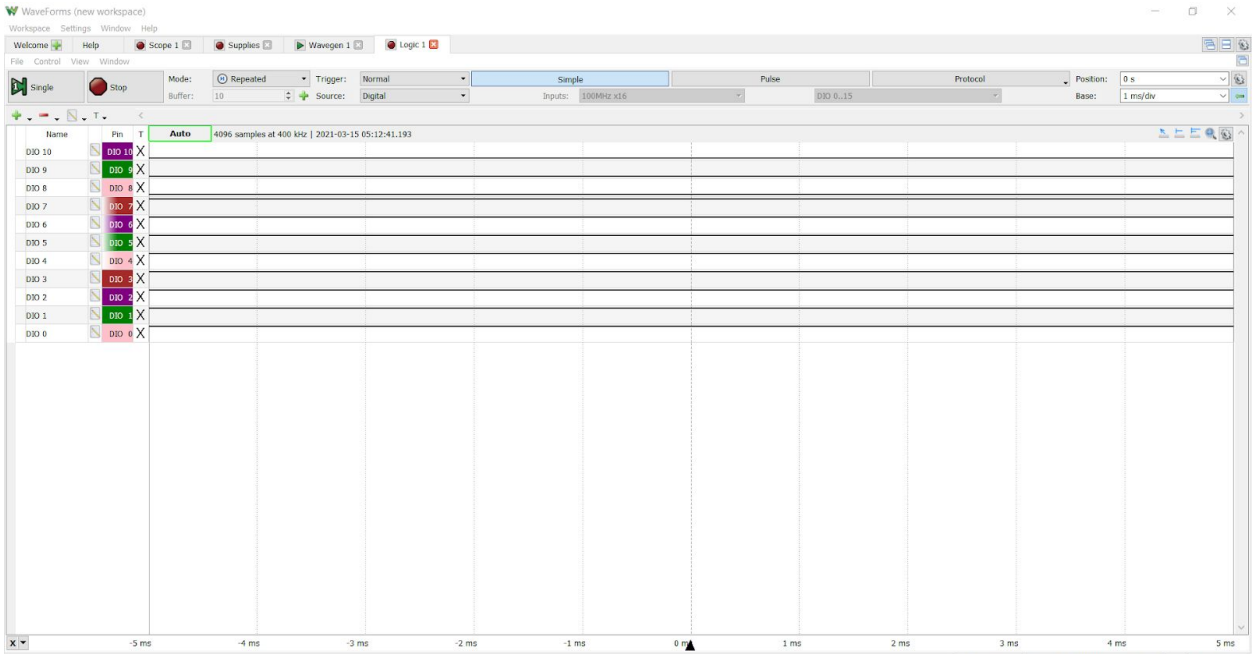


Figure 6.3 - Potentiometer Output Result (Turned to 0xFF)



Conclusion

We were able to accomplish most of the important objectives for this lab. We managed to generate different waveforms (Square, Sine, Triangle, and Ramp) using the lab kit provided by expanding the QC code from Lab 3 and changing it to our software design. We were able to extend the experiment from our Milestones 1 and 2 and continue with our test plans. All the codes are shown in the screenshots above (Figure 1-7). The results for the waveforms (with amplitude and frequency change) are all shown in our “Testing” section above. We ran into trouble where we can’t get the maximum frequency and peak-to-peak that we want, even with changing the number of samples we had, which is most likely due to the quality of the PCB and equipment used.

The final code that we got will be attached with the report as a .pdf file.